Proseslerin Kod, Data, BSS, Stack ve Heap Alanları

Kaan Aslan-Sebahat Ersoy
8 Şubat 2014

1. Giriş

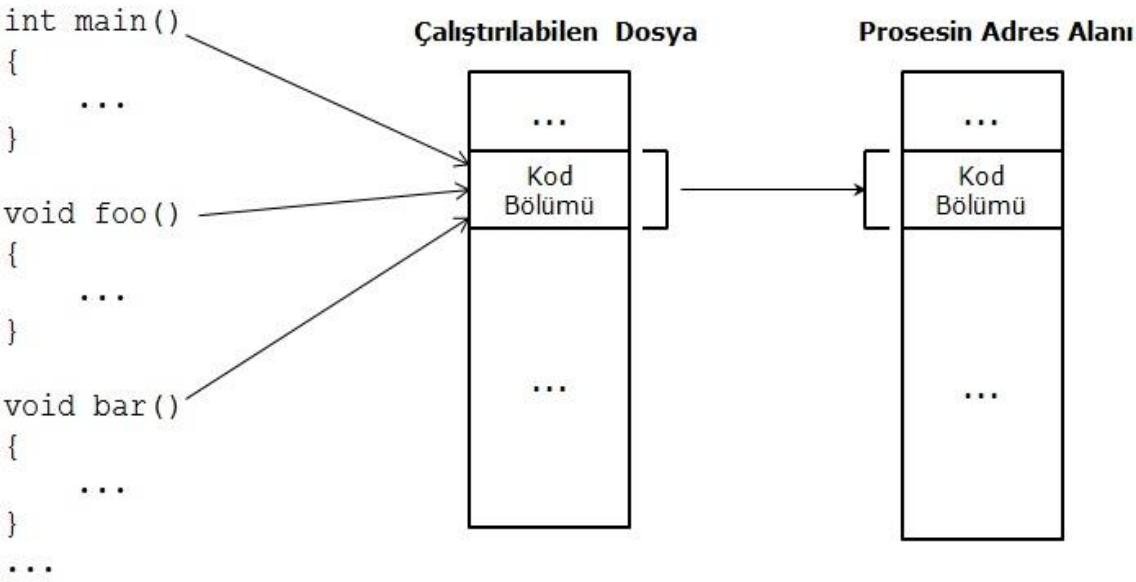
Prosesler işletim sistemlerinin sundukları sistem fonksiyonlarıyla yaratılırlar. Örneğin, *Windows* sistemlerindeki *CreateProcess*, *UNIX/Linux* sistemlerindeki *fork* proses yaratan fonksiyonlardır. *Windows* ve *UNIX/Linux* sistemlerindeki proses yaratımları arasında önemli bir fark vardır. *Windows* sistemlerindeki *CreateProcess* fonksiyonu çalıştırılabilen (*executable*) bir dosyadan hareketle prosesi oluşturur. (Yani yaratılacak proses yol ifadesini argüman olarak verdiğimiz programın kodlarını çalıştıracaktır.) Halbuki *UNIX/Linux* sistemlerindeki *fork* fonksiyonu prosesin özdeş bir kopyasını oluşturmaktadır. Bu sistemlerde başka bir programın kodları ancak *exec* türevi fonksiyonlarla çalıştırılabilir. Fakat programcı bu sistemlerde doğrudan *exec* fonksiyonunu kullanırsa çalışmakta olan mevcut programın bellek alanı değiştirilerek mevcut proses yaşamına başka bir kodla devam eder. Bu nedenle *UNIX/Linux* sisteminde hem mevcut programı sürdürmek hem de yeni bir programı çalıştırmak istiyorsak *fork* ve *exec* fonksiyonlarını beraber kullanmamız gerekir. Bu sağlamak için de, tipik olarak, önce bir kez *fork* yaparız, alt proseste de *exec* uygularız. *Windows* sistemlerindeki *CreateProces* fonksiyonu *UNIX/Linux* sistemlerindeki *fork* ve *exec* fonksiyonlarının bileşimine benzetilebilir.

İşletim sistemlerinin çalıştırılabilen dosyanın içeriğini alarak belleğe yükleyen ve onu çalışmaya hazır hale getiren kısmına yükleyici (*loader*) denilmektedir. *CreateProcess* ve *exec* gibi fonksiyonlarla devreye sokulan yükleyici , önce çalıştırılabilen dosyanın başlık kısmını okur, sonra dosyanın bölümlerinin yerlerini ve uzunluklarını belirleyerek onları belleğe yükler. Sonra da programın akışını çalıştırılabilen dosyanın başlık kısmında belirtilen giriş noktasından (*entry point*) başlatır. Çok çeşitli çalıştırılabilen dosya formatları bulunuyor olsa da iki formatın ağırlığı kendini hissettirmektedir. Bunlardan biri *Micosoft Windows* sistemlerinde kullanılan *PE (Portable Executable)* formatı, diğeri ise *UNIX/Linux* sistemlerinde kullanılan *ELF (Executable and Linkable Format)* formatıdır. Her iki format da başlık kısımlarından ve bölümlerden (*sections*) oluşmaktadır. *PE* ve *ELF* başlıklarında yüklenecek programa ilişkin *metadata* bilgileri (yani bölümlerin yerleri, uzunlukları, programın başlangıç noktası, yeniden yükleme bilgileri vs). tutulmaktadır. Yükleyici başlık kısmından elde ettiği bilgilere dayanarak yükleme işlemini gerçekleştirir.

Yükleyici yükleme işlemi sırasında proses için bellekte çeşitli alanlar oluşturmaktadır. Prosesin bellek alanını oluşturan en önemli bölümler kod, *data*, bss, *stack* ve *heap* bölümleridir. Aşağıda bu bölümlerin işlevleri açıklanmaktadır.

2. Kod Bölümü

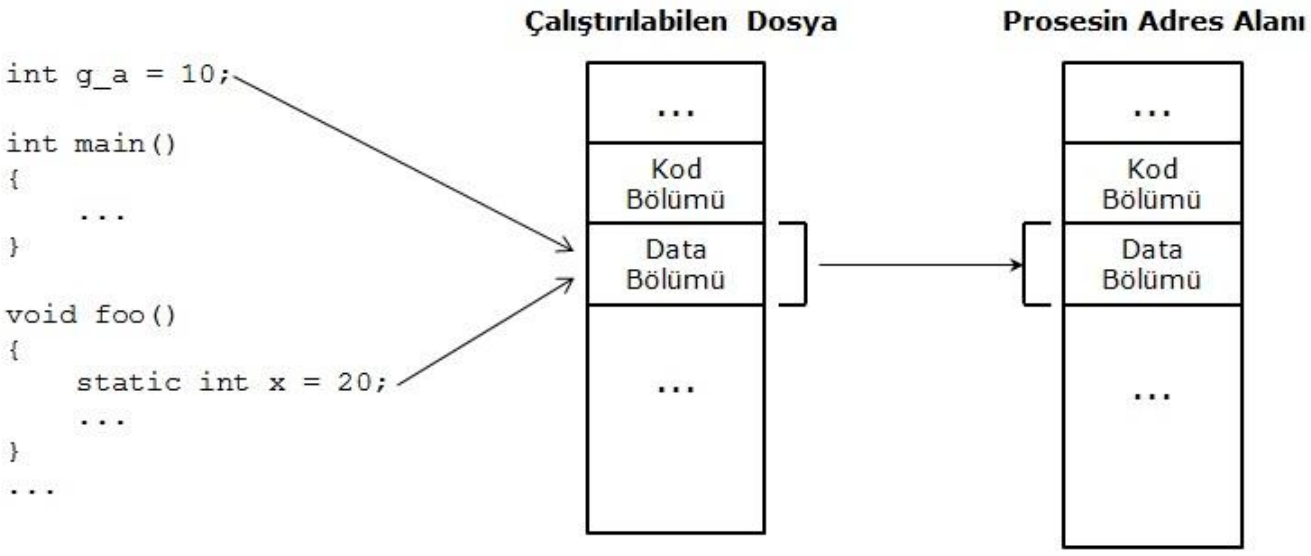
Programı oluşturan tüm fonksiyonların makina kodları derleyici ve bağlayıcının işbirliği ile çalıştırılabilen dosyanın kod bölümüne yazılır.[1] İşletim sisteminin yükleyicisi çalıştırılabilen dosyanın kod bölümünü belleğe blok olarak yükler. Böylece prosesin kod bölümü bellekte oluşturulmuş olur.



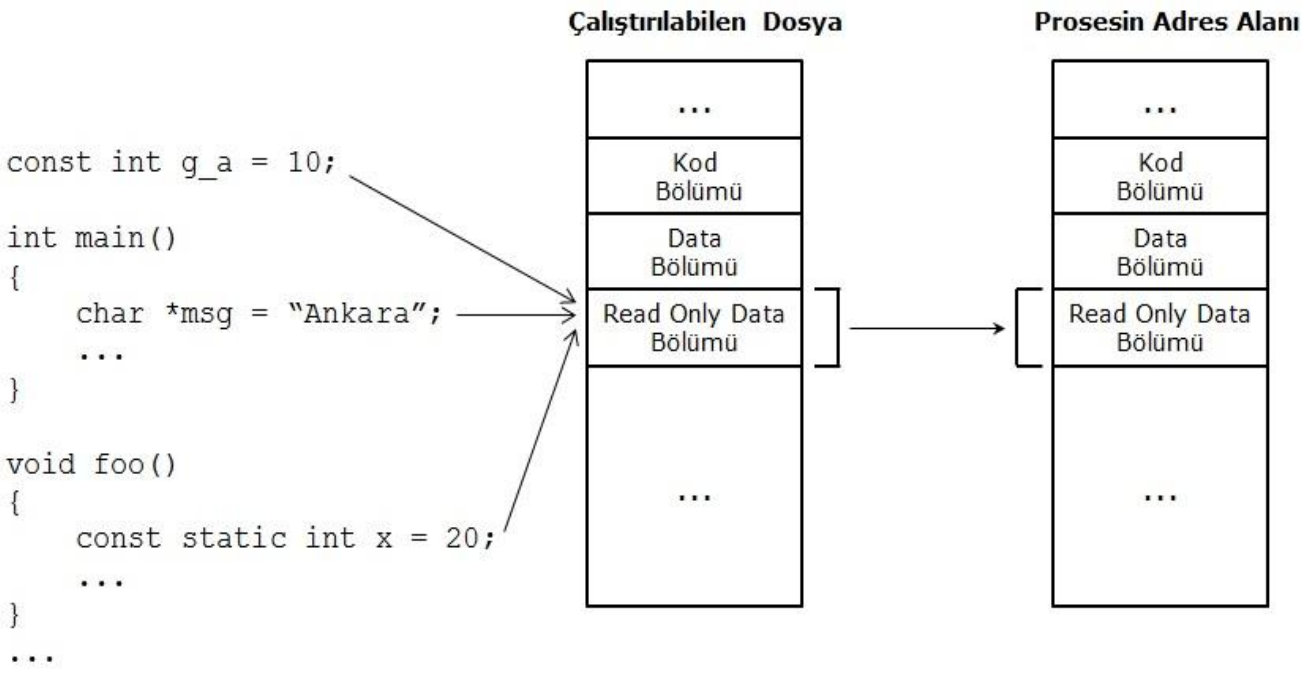
.NET, *Java* gibi ara kodlu çalışma çalışma sisteminin söz konusu olduğu ortamlarda çalıştırılabilen dosyanın kod bölümünde ara kodlar bulunur. Bu ara kodlar belleğe yüklenerek *tam zamanında derleme (JIT Compilation)* işlemiyle gerçek makina komutlarına dönüştürülmektedir.

3. Data ve Read Only Data Bölümleri

Program içerisindeki ilkdeğer verilmiş global değişkenler ve *static* yerel değişkenler ilkdeğerleriyle birlikte derleyici ve bağlayıcının işbirliği ile çalıştırılabilen kodun *data* bölümüne yazılmaktadır. Böylece global ve *static* yerel nesneler programın belleğe yüklenmesiyle hayat kazanmış olurlar.

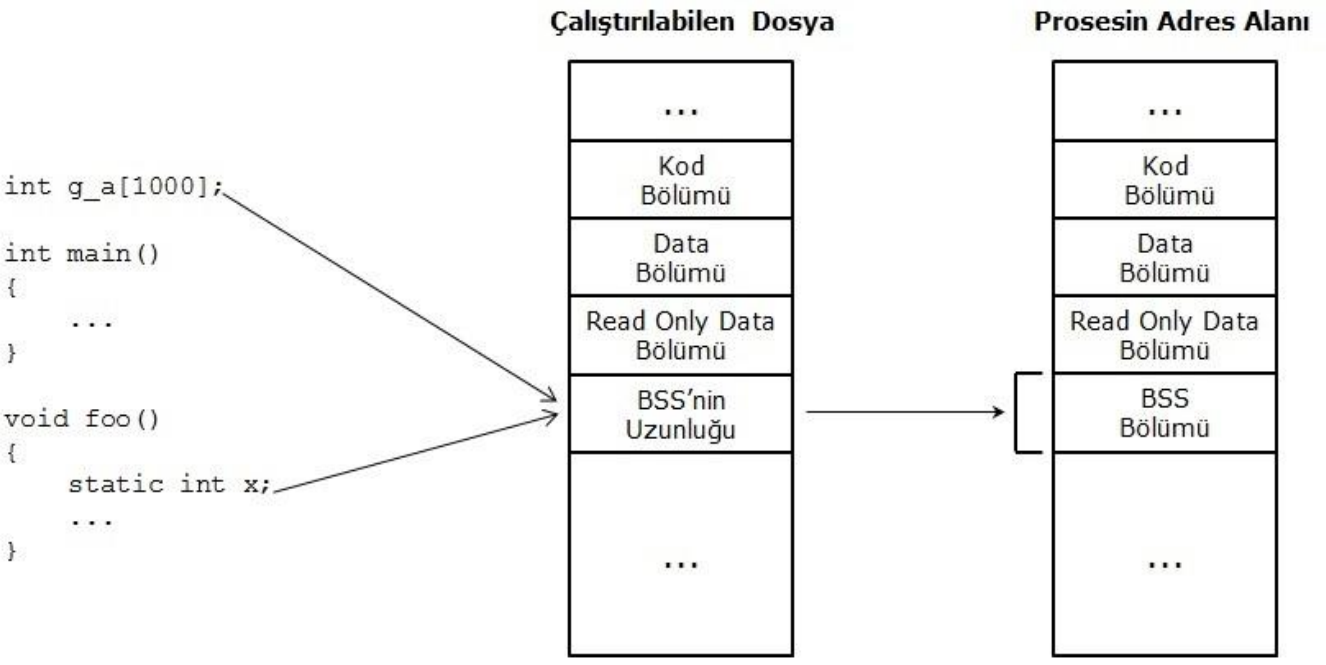


İlkdeğer verilmiş değişkenlerin çalıştırılabilen dosya içerisinde yer kapladığına dikkat ediniz. Pek çok sistemde içeriği değiştirilemeyen (yani *const* olan) *global* ve *static* yerel nesneler ile *string* ifadeleri r*ead only data* bölümünde tutulmaktadır.[3] *Read only data* bölümü yükleyici tarafından belleğin *read only* özelliği verilmiş sayfalarına yüklenirler. Böylece bu nesnelerin programın çalışma zamanı sırasında güncellenmesi engellenmiş olur.[4] Koruma mekanizmasına sahip işlemciler *read only* sayfalara güncelleme yapılmak istendiğinde içsel bir kesmeyle durumu işletim sistemine bildirir, işletim sistemi de hataya yol açan prosesi sonlandırır. *Data* ve *read only data* bölümleri prosesin tüm *thread*'leri tarafından ortak bir biçimde kullanılmaktadır.



4. BSS Bölümü

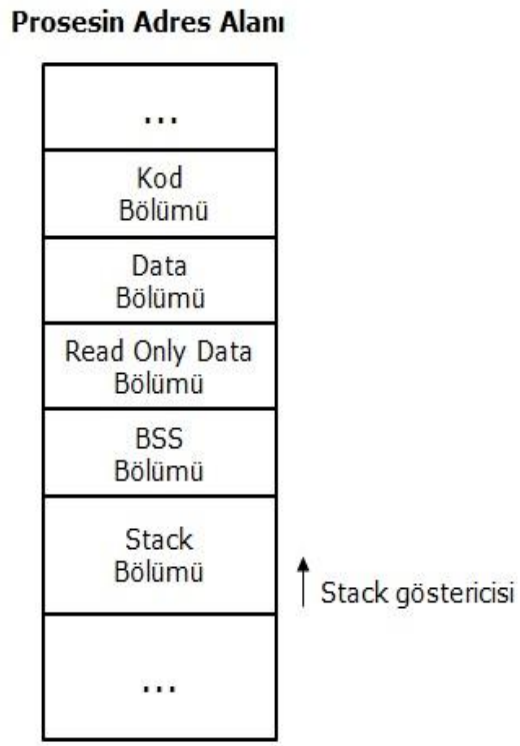
Program içerisindeki ilkdeğer verilmemiş global nesneler ve *static* yerel nesneler çalıştırılabilen dosyanın *bss* bölümünü oluşturmaktadır.[5] *BSS* bölümündeki nesnelere ilkdeğer verilmediği için bunların çalıştırılabilen dosya içerisinde yer kaplamasına gerek yoktur. Bu nedenle *bss* bölümünün yalnızca uzunluğu çalıştırılabilen dosyada belirtilir. Kendisi yükleyici tarafından bellekte oluşturulur. *C* ve *C++*'ta ilkdeğer verilmemiş global ve *static* yerel nesnelerini içerisinde 0 değerinin bulunduğunu anımsayınız. Bu nesnelerin içerisine 0 değerlerini tipik olarak yükleyici atamaktadır. Yani yükleyici çalıştırılabilen dosyadan *bss* alanının uzunluğunu belirler. Bunun için bellekte yer ayırır ve orayı sıfırlar. [6] Ancak bazı sistemlerde (örneğin işletim sisteminin bulunmadığı gömülü sistemlerde), *bss* alanının sıfırlanması henüz akış *main* fonksiyonuna girmeden derleyicilerin başlangıç kodları (*startup codes*) tarafından da yapılabilmektedir.



BSS bölümü de prosesin tüm *thread*'leri arasında ortak bir biçimde kullanılmaktadır.

5. Stack Bölümü

Stack bölümü parametre değişkenlerinin ve yerel değişkenlerin yaratıldığı doldur-boşalt alanıdır. Tipik olarak bir fonksiyon çağrılmadan önce parametre değişkenleri *stack*'e atılır, sonra fonksiyon çağrılır ve fonksiyonun içerisinde yerel değişkenler *stack*'te yaratılır. [7] Fonksiyondan çıkıldığında da yerel değişkenler ve parametre değişkenleri *stack*'ten yok edilirler. *Stack* sisteminin *LIFO* (*Last In First Out*) çalışma prensibinden dolayı yalnızca o ana kadar çağrılmış olan fonksiyonların parametre ve yerel değişkenleri *stack*'te yer kaplamaktadır. Tabii s*tack* bölümünün de çalıştırılabilen dosya içerisinde yer kaplaması için bir neden yoktur. Bu nedenle *stack* bölümünün yalnızca varsayılan uzunluğu çalıştırılabilen formatta belirtilir. *Stack* için yer ayrılması programın yüklenmesi sırasında yükleyici tarafından yapılmaktadır.



Bir program içerisindeki tüm fonksiyonların yerel değişkenleri belli bir anda bellekte yer kaplıyor değildir. Çünkü henüz çağrılmamış fonksiyonların yerel değişkenleri *stack*'te yaratılmamıştır ve çağrılması bitmiş olan fonksiyonların da yerel değişkenleri bellekten yok edilmiştir. (Aksi taktirde bu kadar çok değişken için bellek yetersiz kalabilirdi.) *Stack*'teki yaratımı şuna benzetebiliriz. Bir belediye otobüsü *Sarıyer*'den *Beşiktaş*'a sefer yapıyor olsun. Otobüsteki tüm yolcular *Sarıyer*'den binip *Beşiktaş*'ta inmek zorunda değildir. Bir çok yolcu ara bir durakta binip başka bir durakta inmektedir. Böylece kısıtlı büyüklüğe sahip otobüs yüzlerce kişiyi taşımış olur. Otobüse tüm binmiş olanları biz aynı anda otobüse sığdırabilir miyiz? [8]

Stack'te yerel değişkenler tek bir makine komutuyla çok hızlı bir biçimde yaratılırlar ve yine tek bir makine komutuyla çok hızlı bir biçimde yok edilirler. Bundan dolayı *C* ve *C++* gibi bazı dillerde yerel değişkenlere otomatik değişkenler de denilmektedir. Şimdi de parametre değişkenlerinin ve yerel değişkenlerin *stack*'te nasıl yaratıldıkları üzerinde duralım. İki parametre değişkenine ve üç de yerel değişkene sahip aşağıdaki gibi bir *Foo* fonksiyonu bulunuyor olsun:

```
void Foo(int x, int y)
{
    int a, b, c;
    ...
}
```

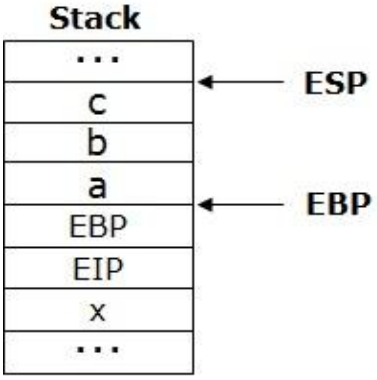
Bunun 32 bit *Intel* işlemcileri için eşdeğer sembolik makine kodları şöyle olabilir:

```
_Foo proc near
    push   ebp
    mov    ebp, esp
    sub    esp, 12
    …
    mov    esp, ebp
    pop    ebp
    ret
_Foo endp
```

Yerel değişkenler *stack* göstericisinin (*ESP*) yukarı doğru çekilmesiyle yaratılmaktadır. Böylece artık *stack* hareketleri yerel değişkenler için ayrılmış alanı etkileyemeyecektir.

```
_Foo proc near
    push ebp
    mov  ebp, esp
    sub  esp, 12      <-- Yerel değişkenler yaratılıyor
    ...
    mov esp, ebp      <-- Yerel değişkenler yok ediliyor
    pop ebp
    ret
_Foo endp
```

Fonksiyonun çağrılması sırasında *stack*'in durumu şöyle olacaktır:



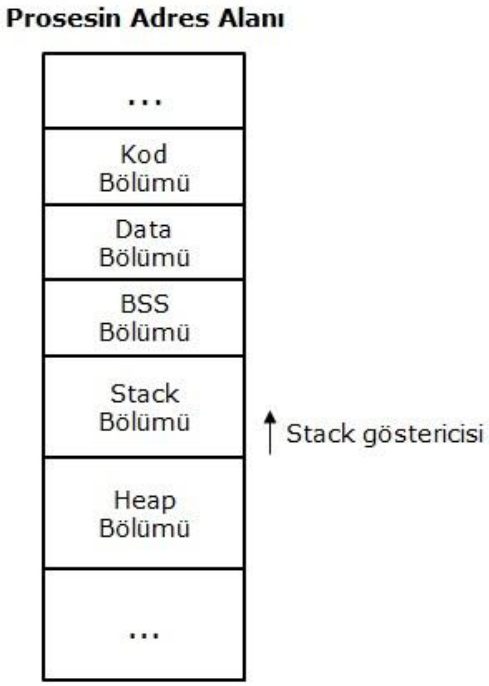
ESP yazmacının 12 *byte* geri çekildiğine dikkat ediniz. 32 bit sistemlerde tipik olarak *int* türü 4 *byte* uzunluktadır.

Çok *thread*'li sistemlerde prosesin her *thread*'i için ayrı bir *stack* yaratılmaktadır. Yani her *thread* kendi *stack*'ine sahiptir. Parametre değişkenleri ve yerel değişkenler *stack*'te yaratıldığı için iki *thread* aynı fonksiyon üzerinde ilerliyor olsa birbirlerinin yerel değişkenlerini bozmazlar. Her *thread*

ayrı bir *stack*'e sahip olduğu için yerel değişkenlerin de ayrı bir kopyasını kullanıyor durumdadır. *Windows* sistemlerinde bir *thread*'in (ana *thread* de dahil olmak üzere) varsayılan uzunluğu *PE* dosyasının *IMAGE_OPTIONAL_HEADER* başlığında belirtilmektedir. *Microsoft* bağlayıcıları buraya varsayılan 1 MB değerini yerleştirirler. (Bu varsayılan *stack* uzunluğu bağlayıcı ayarlarıyla değiştirilebilir.) *Linux* sistemlerinde de *stack* uzunluğunun varsayılan değeri ise 8 MB'dir. Her iki sistemde de varsayılan uzunluk yerine *thread'ler* programcının belirlediği uzunlukla yaratılabilirler.

6. Heap Bölümü

Dinamik bellek fonksiyonları tarafından tahsis edilme potansiyelindeki alanlara *heap* denir. *Heap* alanı da çalıştırılabilen dosyada yer kaplamaz, prosese özgü bir biçimde prosesin adres alanı içerisinde yaratılır. *Heap* alanının prosesin adres alanı içerisinde nerede ve ne kadar uzunlukta yaratılcağı sistemden sisteme değişebilmektedir. Genellikle sistemler prosesin adres alanında başka amaçla kullanılmayan tüm boş yerleri potansiyel bir *heap* alanı olarak belirlemektedir.



Heap üzerinde tahsisatlar bir veri yapısı oluşturularak ve bir tahsisat algoritması kullanılarak gerçekleştirilmektedir. Malloc, calloc gibi *Heap* tahsisat fonksiyonları ortak bir tahsisat tablosuna bakarak çalışırlar. Böylece tahsis edilmiş alanları bilirler ve oraları yeniden tahsis etmeye çalışmazlar. *Heap* sistemini tapu dairesine benzetebilirsiniz. Orada herkesin sahip olduğu arazilerin yerleri ve büyüklükleri tutulmaktadır değil mi?..

Heap yönetimi için tasarlanmış çok çeşitli veri yapıları ve algoritmalar vardır. En yaygın kullanılan tahsisat algoritması boş bağlı liste tekniğidir. Bu teknikte yalnızca boş alanların adresleri ve uzunlukları bir bağlı listede tutulur. *malloc* gibi tahsisat fonksiyonları ile yeni bir alan tahsis edileceği zaman fonksiyon boş alanların bulunduğu bağlı listeye bakarak en uygun boş alanı oradan koparark verir. Programcı alanı *free* gibi bir fonksiyonla boşaltacağı zaman alan yeniden bağlı listeye eklenir. *Heap*'te tahsisatlar göreli biçimde yavaş olma eğilimindedir.

Prosesin *heap* alanı da *data* ve *bss* alanı gibi prosesin tüm *thread*'leri arasında ortak bir biçimde kullanılmaktadır.

7. Bölümlerin Karşılaştırması ve Özet

Prosesin *data*, *read only data*, *bss* ve *heap* alanları tüm *thread*'ler tarafından ortaklaşa kullanılmaktadır. Fakat *thread*'lerin *stack*'leri birbirinden ayrılmıştır. Böylece iki farklı *thread* aynı fonksiyon üzerinde ilerlerken parametre değişkenlerinin ve yerel değişkenlerin kendi *thread*'lerine özgü kopyasını kullanıyor durumda olurlar. Prosesin *stack* alanı küçük olma eğilimindedir. *Data*, *read only data*, *bss* ve *heap* alanları göreli olarak daha büyük olma eğilimindedir. Bu nedenle büyük diziler yerel olarak yaratılmamalı, *static* düzeyde yaratılmalı ya da *heap*'te tahsis edilmelidir. Prosesin kod bölümü fonksiyonların makine komutlarının bulunduğu bölümdür. Programın tüm kodları bu bölümde tutululmaktadır. *Data*, *read only data* ve *bss* bölümleri global ve *static* yerel nesneler için kullanılan bölümdür. İlkdeğer verilerek tanımlanmış global ve *static* yerel nesneler *data* bölümünde, ilkdeğer verilmemiş olanlar *bss* bölümünde bulunurlar. *const* global ve *static* yerel nesneler ve *string* ifadeleri *read only data* bölümünde saklanmaktadır. *Data* ve *read only data* bölümleri çalıştırılabilen dosya içerisinde yer kaplamasına karşın, *bss* bölümü çalıştırılabilen dosya içerisinde yer kaplamaz. *Data*, *read only data* ve *bss* bölümleri programın yüklenmesiyle prosesin adres alanında oluşturulur ve program sonlanana kadar buradaki nesneler bellekte yer kaplamaya devam ederler. *Heap* ise tahsisat tablolarıyla kontrol edilen dinamik alandır. Burada nesneler dinamik olarak tahsis edilirler ve kullanım bittikten sonra geri bırakılabilirler. *Stack* parametre değişkenlerinin ve yerel değişkenlerin yaratıldığı doldur-boşalt alanıdır. Bir fonksiyon çağrıldığında yerel değişkenler *stack*'te yaratılırlar, fonksiyon çalıştığı sürece *stack*'te yer kaplarlar ve fonksiyon sonlandığında da yok edilirler. Hem *stack* hem de *heap* tahsisatları programın çalışma zamanı sırasında yapılmasına karşın, *stack*'teki tahsisat akış fonksiyondan çıktığında yok edilmektedir. Oysa *heap*'te yapılan tahsisatlar onlar serbest bırakılana kadar ya da proses sonlanana kadar kalırlar. *Stack*'te parametre değişkenlerinin ve yerel değişkenlerin yaratılması ve yok edilmesi çok hızlıdır, ancak nesnelerin *heap*'te yaratılması ve yok edilmesi göreli olarak oldukça yavaştır.

Prosesin *data*, *read only data*, *bss* ve *heap* alanları tüm *thread*'ler tarafından ortaklaşa kullanılmaktadır. Fakat *thread*'lerin *stack*'leri birbirinden ayrılmıştır. Böylece iki farklı *thread* aynı fonksiyon üzerinde ilerlerken parametre değişkenlerinin ve yerel değişkenlerin kendi *thread*'lerine özgü kopyasını kullanıyor durumda olurlar. Prosesin *stack* alanı küçük olma eğilimindedir. *Data*, *read only data*, *bss* ve *heap* alanları göreli olarak daha büyük olma eğilimindedir. Bu nedenle büyük diziler yerel olarak yaratılmamalı, *static* düzeyde yaratılmalı ya da *heap*'te tahsis edilmelidir.

Aşağıda prosesin bellek alanlarının çeşitli bakımlardan karşılaştırması tablo haline verilmektedir:

Bölüm	Çalıştırılabilen Dosyada Yer Kaplıyor mu?	Yaratılan Nesnenin Ömürleri	*Thread*'ler Arası Paylaşım	Büyüklük
Kod	Evet	Çalışma zamanı süresince	Evet	Büyük olma eğilminde
Data	Evet	Çalışma zamanı süresinde	Evet	Büyük olma eğilminde
Read Only Data	Evet	Çalışma zamanı süresince	Evet	Büyük olma eğilminde
BSS	Hayır	Çalışma zamanı süresince	Evet	Büyük olma eğilminde
Stack	Hayır	Fonksiyon süresince	Hayır	Küçük olmas eğiliminde
Heap	Hayır	Proses sonlanana kadar ya da alan serbest bırakılana kadar	Evet	Büyük olma eğiliminde

---

[1]: Kod bölümü *PE* ve *ELF* formatlarında *.text* ismiyle temsil edilmektedir.

[2]: Data bölümü *PE* ve *ELF* formatlarında *.data* ismiyle temsil edilmektedir.

3]: *Read only data* bölümü *PE* formatında *.rdata*, *ELF* formatında *.rodata* ismiyle bulunmaktadır.
[4]: *C* ve *C++*'ta *const* nesnelerin ve *string* ifadelerinin değiştirilmesi yani güncellenmesi tanımsız davranışa (*undefined behavior*) yol açmaktadır.

[5]: *BSS* ismi il kez *Roy Nutt* ve *Walter Ramshaw* tarafından *IBM 704* makinaları için geliştirilen sembolik makine dilinde kullanılmıştır. Bu sembolik makine dilinde *BSS* jump için bir etiket oluştururken aynı zamanda belli bir alanı da boş bırakıyormuş.

[6]: *Microsoft*'un *C* ve *C++* derleyicilerinin yeni versiyonları *default* durumda ilkdeğer verilmemiş global ve *static* yerel nesneleri *BSS* bölümüne değil, *Data* bölümünün sonuna yerleştirmektedir. *PE* formatında bir bölümün toplam uzunluğu ile içerijk uzunluğu ayrı ayrı belirtildiği için bu böyle bir yerleşim mümkün olmaktadır. Fakat *bss_seg(…)* isimli pragma komutuyla yine ilkdeğer verilmemiş nesnelerin ayrı bir bölüme yerleştirilmesi sağlanabilir.

[7]: Parametre aktarımı tipik olarak *stack* yoluyla yapılsa da bu durum zorunlu değildir. Aktarım yazmaç yoluyla ya da *Data* alanı yoluyla da yapılabilmektedir.

[8]: Bu örnek *İstanbul* mevcut gerçek durumla tam örtüşmeyebilir. Çünkü bilindiği gibi *İstanbul*'da otobüs ve minibüslere istenildiği kadar yolcu yerleştirilebilmektedir (!).

Kaynaklar

1. Aslan, K. (2002). *UNIX/Linux Sistem Programlama Kurs Notları*. Istanbul: C ve Sistem Programcıları Derneği.

2. Oracle (2011). *Linkers and Libraries Guide.*